VIPER

October-November 1981

Volume 3, number 4 Journal of the VIP Hobby Computer Association
The VIPER was founded by ARESCO, Inc. in June, 1978

****

Contents

VIPHCA INFO 3.04.01

SOFTWARE
Little Loops: VIP-Amation by Tom Swan ... 3.04.02

HARDWARE
CMOS RAMS for your VIP by Brian Hudson 3.04.07

UTILITY
VIP Operating System
for the ELF by Leo F. Hood .. 3.04.08

HARDWARE
Memory Expansion for
the VIP by George Gadbois .. 3.04.13

TUTORIAL
Machine Code: part 3 by Paul Piescik 3.04.14

READER I/O 3.04.20
3.04.21

ANNOUNCEMENTS 3.04.20

BOOK REVIEW:
Tom Swan's Programmer's Guide to the 1802
reviewed by Ray Sills 3.04.22

The VIPER, founded by ARESCO, Inc., in July 1978, is the Official Journal of the VIP Hobby Computer Association. Acknowledgement and appreciation is extended to ARESCO for permission to use the VIPER name. The Association is composed of people interested in the VIP and computers using the 1802 microprocessor. The Association was founded by Raymond C. Sills and created by a Constitution, with By-Laws to govern the operation of the Association. Mr. Sills is serving as Director of the Assocation, as well as editor and publisher of the VIPER.

The VIPER will be published six times per year and sent to all members in good standing. Issues of the VIPER will not carry over from one volume to another. Individual copies of the VIPER and past issues, where they are available, may be sent to interested people for $3 each. Annual dues to the Association, which includes six issues of the VIPER, is $12 per year.

VIP and COSMAC are registered trademarks of RCA Corporation. The VIP Hobby Computer Associatin is in no way associated with RCA, and RCA is not responsible for the contents of this newsletter. Members should not contact RCA regarding material in the VIPER. Please send all inquiries to VIPHCA 32 Ainsworth Avenue, East Brunswick, NJ 08816.

Membership in the VIP Hobby Computer Association is open to all people who desire to promote and enjoy the VIP and other 1802 based systems. Send a check for $12 payable to "VIP Hobby Computer Association" c/o R. C. Sills at the above address. People outside of the United Stated, Canada or Mexico please include $6 extra for postage. All funds must be in U>S. Dollars.

Contributions by members or interested people are welcome at any time. Material submitted by you is assumed to be free of copyright restrictions and will be considered for publication in the VIPER. Articles, letters, programs, etc., in camera-ready form on 8.5 by 11 inch paper will be given preferential consideration. Please send enough information about any program so that the readers can operate the program. Fully documented programs are best, but "memory dumps" are OK if you provide enough information to run the program. Please indicate in your material any key memory locations and data areas.

ADVERTISING RATES

1. Non-commercial classified ads from members, 5 cents per word, minimum of 20 words. Your address or phone number is free.

2. Commercial ads and ads from non-members, 10 cents per work, minimum of 20 words. Your address or phone number is free.

3. Display ads from camera ready copy, $6/half page, $10/page.

Payment in full must accompany all ads. Rates are subject to change.

If you write to VIPER/VIPHCA, please indicate that it is OK to print your address in your letters to the editor, if you want that information released. Otherwise, we will not print your address in the VIPER.

MOVE OVER, MICKEY
HERE COMES VIP-AMATION!

by Tom Swan

I've been doing a lot of experimentation with graphics and animation recently, and last night made a wonderful discovery that vastly improved the graphics I was getting on my VIP. How about a one hundred percent (100%, really!) increase in speed with no, and not just a little, but absolutely no flicker at all with moving objects at full brightness in regular Chip-8 programs with no machine language programming (almost)?

This graphics technique does not rely on "page switching" where smooth animation can be produced by first manipulating unseen bits in memory, then switching to display that area while additional graphics are introduced in the memory area previously shown. By drawing in this way on the unseen memory, and displaying only the final results, flopping back and forth between the pages, absolutely smooth animation is produced. The viewer in this case never sees any changes in progress, only the final and static images.

However, page switching takes extra memory, in fact double the display area of a standard one page display, the kind used in most all Chip-8 programs. I've used this technique successfully, but with memory space at a premium in a 4k VIP, I'd rather keep the memory for other things.

THE CHIP-8 METHOD

Most of you are no doubt familiar with the usual Chip-8 graphics method used to animate balls, tanks, and other objects on the graphics display. The following steps are the ones most often used in Chip-8 games, and I admit, the technique I used extensively in the Pips books and most of the other Vip programs I have written. This process goes something like this:

1) INITIALIZE ALL PROGRAM VARIABLES, VX, VY, ETC
2) DISPLAY OBJECT AT VX,VY TO BEGIN
3) RE-DISPLAY OBJECT AT SAME VX,VY TO ERASE IT
4) CALCULATE NEW VX,VY POSITION
5) IMMEDIATELY DISPLAY OBJECT AGAIN AT NEW VX,VY
6) (continue with program here)
7) REPEAT FROM STEP 3

It would be possible to replace step 5 above with a GOTO 2. However, in a real situation, with other things going on, scoring, sound effects, other graphics, for examples, the above scheme is preferred. Also, re-displaying the moving object at the new X, Y coordinates as soon as possible keeps flicker down.

However it's done, that's the basic Chip-8 graphics idea. Display, erase, move; display, erase, move. The result is a moving, flickering object that usually comes out an annoying, I've always thought, grey color. (Unless your screen is green phosphor in which case the result reminds me too much of way my skin used to look after too much tequila at Bar Paco above the zocalo in Taxco. Are those days really so far away already?)

Let me express the usual Chip-8 method in another way, using a form much like the one in which Pascal programs are written. (Learning Pascal, by the way, has consumed much of my time recently -- anybody else interested out there?) Two things need some explanation. One, the symbol := a colon with an equals sign tail, is the assignment character. It is <u>one</u> symbol, not two separate ones, and can be pronounced 'is assigned the value of' or more simply, 'becomes.' Therefore, the following...

```
VX := 10;
```

...means the variable 'VX' is assigned or given the value 10. The symbol := is used to avoid the ambiguity present when the equals sign is printed alone as it is in the BASIC language, and in comments to Chip-8 programs. For example, does VX = 10 mean that VX is <u>already</u> equal to 10 or does it mean that the program should give that value to the variable? Out of context, the answer isn't clear.

Second, in the following Pascal-like lines, the construction REPEAT/UNTIL is used. This simply means that everything between the REPEAT and the UNTIL is to be repeated until some condition becomes true. There's quite a lot more to the Pascal programming language, but this will do for a short article. I've taken a few liberties, too. (One has to have <u>some</u> fun around here.)

Back to the meat of this piece. Here's the usual Chip-8 graphics method expressed in a sort of Pascal.

```
BEGIN
  DIRECTION := -1;  VX := 16;  VY := 16;
  SHOW FIGURE;
  REPEAT
    ERASE FIGURE;
    VX := VX + DIRECTION;
    SHOW FIGURE;
    IF VX = 0                    { at left edge }
       THEN DIRECTION := +1
       ELSE IF VX = 58           { at right edge }
              THEN DIRECTION := -1
  UNTIL KINGDOMCOME
END.
```

Again, the idea is to erase, move, and show. Here is a Chip-8

program that closely follows this pattern. The program moves an object from left to right at the normal Chip-8 speed. If you want to run the program, it works with the normal unmodified Chip-8 that came in your VIP instruction book.

```
0200    6CFF    ;VC := -1; DIRECTION VARIABLE
  02    6A10    ;VA := 16; SET VX SOMEWHERE ON SCREEN
  04    6B10    ;VB := 16; SET VY SOMEWHERE, TOO
  06    A280    ;I := $0280; POINT TO DISPLAY BIT PATTERN
  08    DAB5    ;SHOW PATTERN AT VX,VY (VA,VB)
  0A    DAB5    ;ERASE PATTERN AT SAME COORDINATES
  0C    8AC4    ;VA := VA + VC; ADD DIRECTION TO VX
  0E    DAB5    ;SHOW PATTERN AT NEW VX, VY (VA,VB)

0210    4A00    ;SKIP IF VA <> 0
  12    6C01    ;  IF VA = 0 THEN DIRECTION (VC) := +1
  14    4A3A    ;SHIP IF VA <> 58
  16    6CFF    ;    ELSE IF VA = 58 THEN DIRECTION (VC) := -1
  18    120A    ;GOTO $020A, REPEATING UNTIL YOUR KINGDOM COMES,
                            OR YOU PULL THE RUN SWITCH DOWN

0280    183C 243C 1800  ;BIT PATTERN FOR DISPLAY
```

The above program took 39 seconds, using the scientific measuring method of looking at my watch and counting, to complete 10 laps across the screen. Also, if you type it in, you will see the grey flicker so familiar in Chip-8 games. If you don't type it in, you will see nothing at all. Aren't these computers wunderfull?

A LESSON FROM UNCLE JIMMY

Cartoons, as all afficionados know, are produced by overlaying cell after cell of slightly altered drawings, photographing each one and displaying the result fast enough to fool our dumb and gullible brains into thinking that such a cute little mouse really can wriggle his ears and power a steamship at the same time as fighting off the big bad wolf. Cellular animation, cellular celluloidics. Whatever. That's how it's done.

What if we did the same thing on a computer display? In other words, instead of erasing the old pattern before redrawing it in a new position, why not simply overlay, just as they do in a cartoon, the old patterns with successive new ones? Instead of having to erase old patterns, the new cells will themselves erase the old images.

The immediate advantage is speed. Because it is no longer necessary to erase the old pattern, one or even several steps can be removed from inside program loops, and those are the first places to look for speed increases, the subject of an earlier Little Loops column.

The disadvantage of this idea is that the objects themselves must be carefully constructed so that while they move to new positions enough of the repositioned images will overlap the old ones and erase them.

One final problem to putting this idea to work. The Chip-8 interpreter is set up to display all patterns via the Exclusive Or (XOR) logic function. This is why a bit displayed on top of another effectively erases that bit. (1 XORed with 1 = 0) Unfortunately, a zero bit, or turned off bit, when displayed on top of one that is already on, does not erase the bit. (0 XORed with 1 = 1) To do that, the DXYN instruction must be modified in the Chip-8 interpreter to make it into a 'copy to screen' instruction instead of a 'combine via XOR with screen' command.

This can be done as shown below. All I am doing with the following is taking out the XOR instructions and replacing them with null operations, setting X=Rc as it already is at these points. Make the following changes to a copy of your Chip-8 interpreter:

1) Change location 00BF from F3 to EC
2) Change location 00CF from F3 to EC

Now the interpreter will allow a zero bit to erase a one bit. The following program and Pascal-like algorithm will demonstrate the change. Because the Chip-8 inner display loop is shorter, the same program as before took just under 20 seconds to complete 10 laps across the screen, approximately twice as fast as before. The speed increase has absolutely nothing to do with the changes made to the interpreter. Also, and I think this will please you, absolutely no flicker can be seen as the object travels. Furthermore, the object remains at full brightness, the same as an object standing still. I'd be most interested in seeing some games developed using this idea and published in the Viper. Some additonal experimentation will be needed to use this method in a complete game. Without the XOR function in the interpreter, you cannot erase things by redisplaying them even if you want to. One possible solution would be to let the program modify the interpreter at appropriate times. I leave the rest to your imaginations.

Best of luck to everyone!

```
BEGIN
   DIRECTION := +1;  VX := 16;  VY := 16;
   REPEAT
      SHOW FIGURE;
      VX := VX + DIRECTION;
      IF VX = 0
         THEN DIRECTION := -1
         ELSE IF VX = 58
                 THEN DIRECTION := +1
   UNTIL THE COWS COME HOME
END.
```

{CHIP-8 'COPY TO SCREEN' FAST GRAPHICS PROGRAM -- BE SURE TO MAKE THE TWO MODIFICATIONS TO THE INTERPRETER AS DESCRIBED IN THE ABOVE ARTICLE.}

```
0200    6CFF    ;VC := -1; SET DIRECTION TO MOVE LEFT
  02    6A10    ;VA := 16; SET VX TO SOME SCREEN LOCATION
  04    6B10    ;VB := 16; SET VY TO SOME SCREEN LOCATION, TOO
  06    A280    ;I := $0280; POINT TO DISPLAY BIT PATTERN
  08    DAB5    ;COPY PATTERN TO SCREEN AT VX,VY (VA,VB)
  0A    8AC4    ;VA := VA + VC; ADD DIRECTION TO VX
  0C    4A00    ;SKIP IF VA <> 00
  0E    6C01    ;  IF VA = 0 THEN VC := +1

0210    4A3A    ;SKIP IF VA <> 58
  12    6CFF    ;    ELSE IF VA = 53 THEN VC := -1
  14    1208    ;REPEAT UNTIL YOU HEAR MOO'S COMING UP THE PATH

0280    183C 243C 1800  ;BIT PATTERN FOR DISPLAY
```

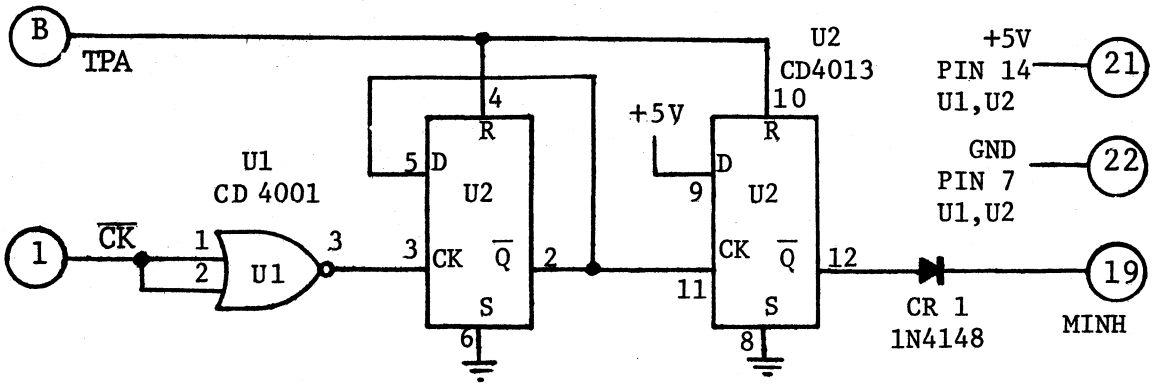
```
**      END     **      END     **      END     **
```

CMOS RAMS FOR YOUR VIP

by Brian Hudson

If you have considered adding four 2114 RAMs to your basic VIP, but worry about overloading the stock power supply, or if you want to run your VIP on batteries, then switch to CMOS RAMs! The 6514 RAM offered by EMERGE SYSTEMS in VIPER 2.02.28 and the MWS5114 available from RCA have the same pinout as the 2114, and will replace the 2114 in most applications at a fraction of the power supply current. The CMOS RAMs use internal address latches that hold the address present when the Chip Select (CS) goes low, and this causes a problem with the VIP's multiplexed address buss. When CS goes low during the VIP's memory read (MRD) cycle, the high-order address byte is still on the address buss, and the CMOS RAMs would latch the wrong address. But a simple plug-in circuit can solve this problem by delaying the CS until the low-order address byte from the CPU is on the address buss. Just wire the two ICs on a 44-pin connector perf. board, and plug it into the VIP's expansion interface connector (left side). Plug four CMOS RAMs into the sockets provided for U20-23. The delayed CS does not affect the operation of the 2114 RAMs. The board will even work with the System Expansion Board VP-575. If you use the EPROM Board VP-560, a diode must be inserted in the line going to Pin 19, as in this schematic, to prevent conflict between that board's decoder for the VIP's on-board RAM and this modification. RCA used this circuit in the earlier VIPs, since the AMD9131 RAMs also have address latches, but it was not required in the later VIPs with 2114 RAMs.

For battery-powered applications, carefully unsolder the four 2114 RAMs U16-19, and install 18-pin sockets for the CMOS RAMs. Power supply current can be reduced from 250mA to only 50mA. Replacing the 7474 dual flip-flop at U4 with a 74LS74 saves 12mA, and disconnecting the power-on LED will save another 6mA for truly low-power operation!



VIP Operating System for ELF

by

Leo F. Hood

```
1000  90      GHI  R0
1001  B2      PHI  R2
1002  F806    LDI  #06
1004  A2      PLO  R2
1005  D2      SEP  R2
1006  E2      SEX  R2
1007  67      OUT   7
1008  0C      LDN  RC
1009  F80F    LDI  #0F
100B  B1      PHI  R1
100C  BB      PHI  RB
100D  FF01    SMI  #01
100F  B6      PHI  R6
1010  F8FF    LDI  #FF
1012  A1      PLO  R1
1013  3628    B3   #28
1015  67      OUT   7
1016  0E      LDN  RE
1017  F800    LDI  #00
1019  A0      PLO  R0
101A  B0      PHI  R0
101B  3622    B3   #22
101D  F803    LDI  #03
101F  A0      PLO  R0
1020  E0      SEX  R0
1021  D0      SEP  R0
1022  F8F0    LDI  #F0
1024  3020    BR   #20
1026  E1      SEX  R1
1027  E1      SEX  R1
1028  E1      SEX  R1
1029  F800    LDI  #00
102B  73      STXD
102C  81      GLO  R1
102D  FBAF    XRI  #AF
102F  3A29    BNZ  #29
1031  F8D2    LDI  #D2
1033  73      STXD
1034  F89F    LDI  #9F
1036  51      STR  R1
1037  81      GLO  R1
1038  A0      PLO  R0
1039  91      GHI  R1
103A  B0      PHI  R0
103B  F8CF    LDI  #CF
103D  A1      PLO  R1
103E  D0      SEP  R0
103F  73      STXD

1040  20      DEC  R0
1041  20      DEC  R0
1042  40      LDA  R0
1043  FF01    SMI  #01
1045  20      DEC  R0
1046  50      STR  R0
1047  FB82    XRI  #82
1049  3A3E    BNZ  #3E
104B  92      GHI  R2
104C  B3      PHI  R3
104D  F851    LDI  #51
104F  A3      PLO  R3
1050  D3      SEP  R3
1051  90      GHI  R0
1052  B2      PHI  R2
1053  BB      PHI  RB
1054  BD      PHI  RD
1055  F811    LDI  #11
1057  B1      PHI  R1
1058  B4      PHI  R4
1059  B5      PHI  R5
105A  B7      PHI  R7
105B  BA      PHI  RA
105C  BC      PHI  RC
105D  F846    LDI  #46
105F  A1      PLO  R1
1060  F8AF    LDI  #AF
1062  A2      PLO  R2
1063  F8DD    LDI  #DD
1065  A4      PLO  R4
1066  F8C6    LDI  #C6
1068  A5      PLO  R5
1069  F8BA    LDI  #BA
106B  A7      PLO  R7
106C  F8A1    LDI  #A1
106E  AC      PLO  RC
106F  E2      SEX  R2
1070  69      INP   1
1071  DC      SEP  RC
1072  D7      SEP  R7
1073  D7      SEP  R7
1074  D7      SEP  R7
1075  B6      PHI  R6
1076  D7      SEP  R7
1077  D7      SEP  R7
1078  D7      SEP  R7
1079  A6      PLO  R6
107A  D4      SEP  R4
107B  DC      SEP  RC
107C  BE      PHI  RE
107D  32F4    BZ   #F4
107F  FB0A    XRI  #0A
```

```
1081  32EF  BZ   #EF
1083  DC    SEP  RC
1084  AE    PLO  RE
1085  22    DEC  R2
1086  61    OUT   1
1087  9E    GHI  RE
1088  FB0B  XRI  #0B
108A  32C2  BZ   #C2
108C  9E    GHI  RE
108D  FB0F  XRI  #0F
108F  3A8F  BNZ  #8F
1091  F86F  LDI  #6F
1093  AC    PLO  RC
1094  F880  LDI  #80
1096  B9    PHI  R9
1097  93    GHI  R3
1098  F6    SHR
1099  DC    SEP  RC
109A  29    DEC  R9
109B  99    GHI  R9
109C  3A97  BNZ  #97
109E  F810  LDI  #10
10A0  A7    PLO  R7
10A1  F808  LDI  #08
10A3  A9    PLO  R9
10A4  46    LDA  R6
10A5  B7    PHI  R7
10A6  9E    GHI  RE
10A7  F6    SHR
10A8  DC    SEP  RC
10A9  86    GLO  R6
10AA  3AAD  BNZ  #AD
10AC  2E    DEC  RE
10AD  97    GHI  R7
10AE  F6    SHR
10AF  B7    PHI  R7
10B0  DC    SEP  RC
10B1  29    DEC  R9
10B2  89    GLO  R9
10B3  3AAD  BNZ  #AD
10B5  17    INC  R7
10B6  87    GLO  R7
10B7  F6    SHR
10B8  DC    SEP  RC
10B9  8E    GLO  RE
10BA  3A9E  BNZ  #9E
10BC  DC    SEP  RC
10BD  69    INP   1
10BE  26    DEC  R6
10BF  D4    SEP  R4

10C0  30C0  BR   #C0
10C2  F883  LDI  #83
10C4  AC    PLO  RC
10C5  F80A  LDI  #0A
10C7  B9    PHI  R9
10C8  DC    SEP  RC
10C9  33C5  BDF  #C5
10CB  29    DEC  R9
10CC  99    GHI  R9
10CD  3AC8  BNZ  #C8
10CF  DC    SEP  RC
10D0  3BCF  BNF  #CF
10D2  F809  LDI  #09
10D4  A9    PLO  R9
10D5  A7    PLO  R7
10D6  97    GHI  R7
10D7  76    SHRC
10D8  B7    PHI  R7
10D9  29    DEC  R9
10DA  DC    SEP  RC
10DB  89    GLO  R9
10DC  3AD6  BNZ  #D6
10DE  87    GLO  R7
10DF  F6    SHR
10E0  33E3  BDF  #E3
10E2  7B    SEQ
10E3  97    GHI  R7
10E4  56    STR  R6
10E5  16    INC  R6
10E6  86    GLO  R6
10E7  3ACF  BNZ  #CF
10E9  2E    DEC  RE
10EA  8E    GLO  RE
10EB  3ACF  BNZ  #CF
10ED  30BD  BR   #BD
10EF  DC    SEP  RC
10F0  16    INC  R6
10F1  D4    SEP  R4
10F2  30EF  BR   #EF
10F4  D7    SEP  R7
10F5  D7    SEP  R7
10F6  D7    SEP  R7
10F7  56    STR  R6
10F8  D4    SEP  R4
10F9  16    INC  R6
10FA  30F4  BR   #F4
10FC  00    IDL
10FD  00    IDL
10FE  00    IDL
10FF  00    IDL
```

```
1100  3039    BR   #39
1102  22      DEC  R2
1103  2A      DEC  RA
1104  3E20    BN3  #20
1106  24      DEC  R4
1107  3426    B1   #26
1109  28      DEC  R8
110A  2E      DEC  RE
110B  18      INC  R8
110C  14      INC  R4
110D  1C      INC  RC
110E  10      INC  R0
110F  12      INC  R2
1110  F0      LDX
1111  80      GLO  R0
1112  F0      LDX
1113  80      GLO  R0
1114  F0      LDX
1115  80      GLO  R0
1116  80      GLO  R0
1117  80      GLO  R0
1118  F0      LDX
1119  50      STR  R0
111A  70      RET
111B  50      STR  R0
111C  F0      LDX
111D  50      STR  R0
111E  50      STR  R0
111F  50      STR  R0
1120  F0      LDX
1121  80      GLO  R0
1122  F0      LDX
1123  10      INC  R0
1124  F0      LDX
1125  80      GLO  R0
1126  F0      LDX
1127  90      GHI  R0
1128  F0      LDX
1129  90      GHI  R0
112A  F0      LDX
112B  10      INC  R0
112C  F0      LDX
112D  10      INC  R0
112E  F0      LDX
112F  90      GHI  R0
1130  F0      LDX
1131  90      GHI  R0
1132  90      GHI  R0
1133  90      GHI  R0
1134  F0      LDX
1135  10      INC  R0
1136  10      INC  R0
1137  10      INC  R0
1138  10      INC  R0
1139  60      IRX

113A  20      DEC  R0
113B  20      DEC  R0
113C  20      DEC  R0
113D  70      RET
113E  A0      PLO  R0
113F  A0      PLO  R0
1140  F0      LDX
1141  20      DEC  R0
1142  20      DEC  R0
1143  7A      REQ
1144  42      LDA  R2
1145  70      RET
1146  22      DEC  R2
1147  78      SAV
1148  22      DEC  R2
1149  52      STR  R2
114A  C4      NOP
114B  19      INC  R9
114C  F800    LDI  #00
114E  A0      PLO  R0
114F  9B      GHI  RB
1150  B0      PHI  R0
1151  E2      SEX  R2
1152  E2      SEX  R2
1153  80      GLO  R0
1154  E2      SEX  R2
1155  E2      SEX  R2
1156  20      DEC  R0
1157  A0      PLO  R0
1158  E2      SEX  R2
1159  20      DEC  R0
115A  A0      PLO  R0
115B  E2      SEX  R2
115C  20      DEC  R0
115D  A0      PLO  R0
115E  3C53    BN1  #53
1160  98      GHI  R8
1161  3267    BZ   #67
1163  AB      PLO  RB
1164  2B      DEC  RB
1165  8B      GLO  RB
1166  B8      PHI  R8
1167  88      GLO  R8
1168  3243    BZ   #43
116A  7B      SEQ
116B  28      DEC  R8
116C  3044    BR   #44
116E  D3      SEP  R3
116F  F80A    LDI  #0A
1171  3B76    BNF  #76
1173  F820    LDI  #20
1175  17      INC  R7
1176  7B      SEQ
1177  BF      PHI  RF
1178  FF01    SMI  #01
117A  3A78    BNZ  #78
117C  396E    BNQ  #6E
117E  7A      REQ
117F  9F      GHI  RF
```

```
1180  3078      BR   #78
1182  D3        SEP  R3
1183  F810      LDI  #10
1185  3585      B2   #85
1187  358F      B2   #8F
1189  FF01      SMI  #01
118B  3A87      BNZ  #87
118D  17        INC  R7
118E  8C        GLO  RC
118F  FE        SHL
1190  3D90      BN2  #90
1192  3082      BR   #82
1194  D3        SEP  R3
1195  E2        SEX  R2
1196  9C        GHI  RC
1197  AF        PLO  RF
1198  2F        DEC  RF
1199  22        DEC  R2
119A  8F        GLO  RF
119B  52        STR  R2
119C  62        OUT   2
119D  E2        SEX  R2
119E  E2        SEX  R2
119F  3E98      BN3  #98
11A1  F804      LDI  #04
11A3  A8        PLO  R8
11A4  88        GLO  R8
11A5  3AA4      BNZ  #A4
11A7  F804      LDI  #04
11A9  A8        PLO  R8
11AA  36A7      B3   #A7
11AC  88        GLO  R8
11AD  31AA      BQ   #AA
11AF  8F        GLO  RF
11B0  FA0F      ANI  #0F
11B2  52        STR  R2
11B3  3094      BR   #94
11B5  00        IDL
11B6  00        IDL
11B7  00        IDL
11B8  00        IDL
11B9  D3        SEP  R3
11BA  DC        SEP  RC
11BB  FE        SHL
11BC  FE        SHL
11BD  FE        SHL
11BE  FE        SHL
11BF  AE        PLO  RE

11C0  DC        SEP  RC
11C1  8E        GLO  RE
11C2  F1        OR
11C3  30B9      BR   #B9
11C5  D4        SEP  R4
11C6  AA        PLO  RA
11C7  0A        LDN  RA
11C8  AA        PLO  RA
11C9  F805      LDI  #05
11CB  AF        PLO  RF
11CC  4A        LDA  RA
11CD  5D        STR  RD
11CE  8D        GLO  RD
11CF  FC08      ADI  #08
11D1  AD        PLO  RD
11D2  2F        DEC  RF
11D3  8F        GLO  RF
11D4  3ACC      BNZ  #CC
11D6  8D        GLO  RD
11D7  FCD9      ADI  #D9
11D9  AD        PLO  RD
11DA  30C5      BR   #C5
11DC  D3        SEP  R3
11DD  22        DEC  R2
11DE  06        LDN  R6
11DF  73        STXD
11E0  86        GLO  R6
11E1  73        STXD
11E2  96        GHI  R6
11E3  52        STR  R2
11E4  F806      LDI  #06
11E6  AE        PLO  RE
11E7  F8D8      LDI  #D8
11E9  AD        PLO  RD
11EA  02        LDN  R2
11EB  F6        SHR
11EC  F6        SHR
11ED  F6        SHR
11EE  F6        SHR
11EF  D5        SEP  R5
11F0  42        LDA  R2
11F1  FA0F      ANI  #0F
11F3  D5        SEP  R5
11F4  8E        GLO  RE
11F5  F6        SHR
11F6  AE        PLO  RE
11F7  32DC      BZ   #DC
11F9  3BEA      BNF  #EA
11FB  1D        INC  RD
11FC  1D        INC  RD
11FD  30EA      BR   #EA
11FF  01        LDN  R1
```

VIP ROM Address Modification

from the RCA-Findlay group

The VIP can address only 32K of memory because of the ROM enabling circuit. The ROM is enabled at any address over 8000 hex. Since the processor is capable of addressing twice this amount of memory, and since come commercially available periferals and software use this "high order" address region, you may wish to modify your VIP to use the upper 32K. This modification will cause the ROM address to be limited to 1K of memory (8000-83FF).

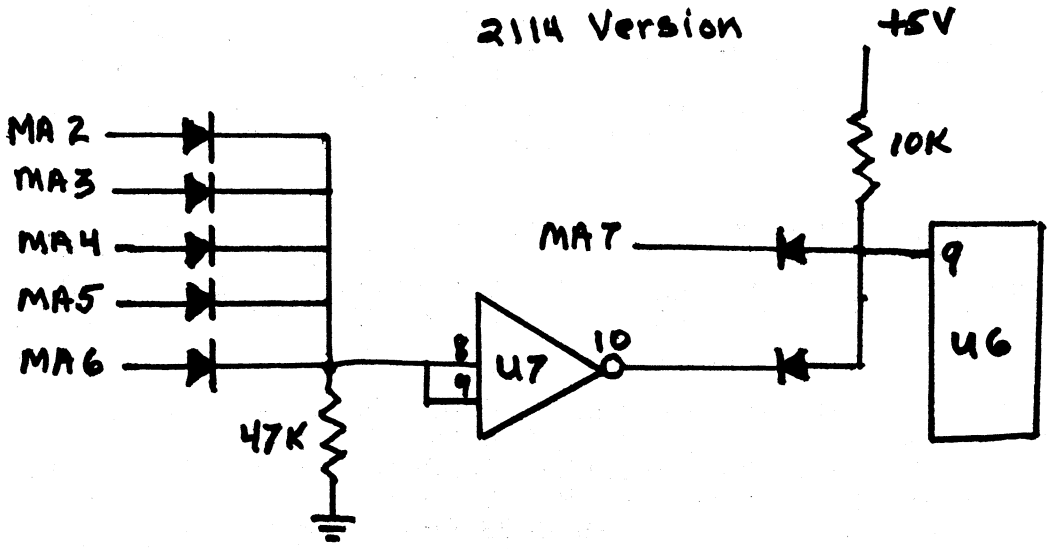
Parts needed: one 47K 1/8 watt resistor
one 10K 1/8 watt resistor
seven 1N914 (or equivalent) diodes

Note: If your VIP uses the AM9131 memory chips, it will be necessary to add your own inverter IC in place of U7. In the 2114 version VIP, there is an unused gate in U7 that becomes an inverter when both inputs are tied together.

Directions for the 2114 version

Cut the land from U6 pin 9 on the top of the PC board (component side). Place one of the AND gate diodes across the cut, anode end to pin 9. Connect the 10K resistor between +5V and pin 9. Connect another diode (anode end) to pin 9 of U6 and the cathode to U7 pin 10. Connect all cathode ends of the remaining 5 diodes together along with the 47K resistor.

Connect the common point of all these components to pins 8 and 9 of U7. Note: Pins 8 & 9 must be disconnected from ground or +5V first. The other end of the 47K resistor must be conected to ground at U7 pin 7. Connect the remaining diode anodes to U23, pins 15, 5, 2, 3, and 4.



VIP MEMORY EXPANSION

BY GEORGE GADBOIS, W3FEY

UNLESS YOU HAVE SOME SPECIAL DESIRE TO PROVE THAT YOU KNOW HOW TO DESIGN MEMORY CIRCUITS, I SUGGEST THAT YOU START BY GETTING A COPY OF RCA PUBLICATION SSD-260, "COS/MOS MEMORIES, MICROPROCESSORS, AND SUPPORT SYSTEMS". THIS BOOK CONTAINS TECHNICAL DATA ON 1802 SUPPORT CHIPS AND ON THE RCA MICROBOARD COMPUTERS. OF SPECIAL INTEREST ARE THE CIRCUIT DIAGRAMS FOR THE MICROBOARD COMPUTERS INCLUDING MEMORY AND I/O BOARDS.

ADDING MEMORY TO THE VIP REQUIRES SOME SPECIAL TREATMENT BECAUSE THE ON CARD RAM ADDRESSES ARE NOT FULLY DECODED. THE RESULT IS THAT THE ON CARD RAM IS 'MIRRORED' IN EVERY 4K BLOCK OF RAM IN THE FIRST 32K OF RAM ADDRESSES. WHAT THIS MEANS IS THAT IF YOU SELECT MEMORY ADDRESS 1000, 2000,....,7000 YOU WILL ACTUALLY GET MEMORY LOCATION 0000. SIMILARLY, 1001,1002, ... ETC. ARE EQUIVALENT TO 0001,0002,...., ETC. WHEN ADDRESSES ABOVE 8000 ARE USED THE ON CARD RAM IS INHIBITED AND YOU WILL GET THE OPERATING SYSTEM ROM. HEREIN LIES THE KEY TO ADDING MEMORY TO THE VIP. BY PULLING THE MEMORY INHIBIT LINE(MINH) LINE HIGH THE ON CARD RAM IS DESELECTED. THUS, WHEN RAM IS ADDED TO THE VIP THE BLOCK SELECT SIGNAL FOR THE ADD ON RAM MUST BE USED TO INHIBIT THE ON CARD RAM VIA THE MINH LINE(PIN 19 ON THE EXPANSION INTERFACE). DIODE ISOLATION MUST BE USED ON THE BLOCK SELECT LINE TO AVOID PROBLEMS WHEN MORE THAN ONE ADD ON RAM CARD IS USED.

THERE IS NO NEED TO DESIGN YOUR OWN MEMORY CIRCUITS. RCA HAS ALREADY DONE THAT FOR YOU AND PUBLISHED THE RESULTS IN SSD-260. TAKE THE 16K MICROBOARD RAM CARD, 18S621, AS AN EXAMPLE. ALL YOU NEED TO DO TO USE THIS CIRCUIT WITH THE VIP IS TO REARRANGE THE EDGE CONNECTOR TO BE COMPATIBLE WITH THE VIP AND BRING THE BLOCK SELECT SIGNAL BACK TO THE MINH LINE(PIN 19). THIS IS EASILY ACCOMPLISHED BY BUFFERING THE BLOCK SELECT SIGNAL WITH THE SPARE BUFFER SECTION IN U6A AND FEEDING THE SIGNAL THROUGH AN ISOLATION DIODE TO THE MINH LINE OF THE VIP. (NOTE THAT R1, R2, R3, AND R4 ARE PULL DOWN RESISTORS NOT PULL UP RESISTORS AS SHOWN ON THE CIRCUIT DIAGRAM.) I WOULD SUGGEST CONNECTING THE VIP 8000 LINE (PIN X ON THE EXPANSION INTERFACE) TO THE RNU (RUN UTILITY) PIN ON THE 16K RAM CARD. THIS WILL AVOID PROBLEMS IF YOU INADVERTENTLY SET THE ADDRESS BLOCK FOR THE ADD ON RAM ABOVE 8000.

REMEMBER THAT THE VIP ON CARD RAM WILL SHOW UP IN EVERY 4K BLOCK IN THE FIRST 32K UNLESS INHIBITED. THUS IF YOUR ADD ON RAM IS AT 0000-3FFF, THE ON CARD RAM WILL APPEAR AT 4000,5000,...., ETC. IF YOU WANT TO ADD MORE THAN 32K OF RAM TO YOUR VIP, IT WILL BE NECESSARY TO CHANGE THE ADDRESS DECODING FOR THE OPERATING SYSTEM ROM. MOST PEOPLE WHO DO THIS CHANGE THE OPERATING SYSTEM TO BE MORE CONVENIENT WITH THE LARGER MEMORY AND MOVE IT OFF THE MAIN CARD.

THE RCA MICROBOARD MEMORIES USE CMOS RAM WHICH SAVES DRAMATICALLY ON POWER CONSUMPTION BUT THEY ARE RATHER EXPENSIVE. INCIDENTALLY YOU CAN USE ONE OF THE BATTERY BACK UP RAM CARDS: WRITE A PROGRAM TO THE CARD AND THEN PULL THE CARD AND MAIL THE PROGRAM TO A FRIEND. TRY THAT WITH NMOS!

I HAVEN'T TRIED USING 2114S IN ANY OF THE MICROBOARD CIRCUITS BUT I DON'T SEE WHY YOU CAN'T JUST SUBSTITUTE 2114S FOR 5114S. LOW POWER 2114 CHIPS ARE READILY AVAILABLE THROUGH RELIABLE SUPPLIERS FOR JUST OVER $3.00 EACH. I HAVE USED NEC 2114S IN MY VIP WITH A SIGNIFICANT REDUCTION IN POWER CONSUMPTION. SOMETIMES CMOS RAM CHIPS SHOW UP IN FLEA MARKETS FOR AROUND $10 WHICH IS PROBABLY A GOOD BUY. I HAVE SEEN 6514'S ADVERTISED AT $7.00. SEE VIPER 2.06.02 FOR USAGE INFORMATION.

MACHINE CODE

P. V. Piescik, 157 Charter Rd., Wethersfield, CT 06109

After re-reading my last column to see what it looks like when I'm not in the middle of writing it, I think I short-changed you. With the end of my page-allotment coming up and a lot of material to go, I stepped up the pace to the point where I got lost, frustrated, and gave up! With a promise to watch that in the future, I'll back up a bit now, before getting into new material.

Let's put all the code together, since mentally combining several pieces of code is difficult at best, and I doubt that you picked up a pencil!

```
     0000  F8 00            LDI 0                ..this page
     0002  B2                PHI 2
     0003  F8 FF            LDI #FF              ..last byte
     0005  A2                PLO 2                ..R2=00FF
     0006  E2                SEX 2                ..X=2, stack ready
     0007  xx          ..next available location

     0080                ADD: ORG *               ..new entry point
     0080  8E                   GLO E             ..D=RE.0
(1)  0081  73                   STXD              ..M(R2)=D, R2=00FE
     0082  9E                   GHI E             ..D=RE.1
(2)  0083  73                   STXD              ..M(R2)=D, R2=00FD
     0084                ..old entry point w/o register save
     0084  F8 00 BE             LDI A.1(ONE);PHI E
     0087  F8 97 AE             LDI A.0(ONE);PLO E   ..RE=A(ONE)
     008A  EE                   SEX E             ..X=E
     008B  4E                   LDA E             ..D=(ONE), RE=A(TWO)
     008C  F4                   ADD               ..D=(ONE)+(TWO)
     008D  1E                   INC E             ..RE=A(SUM)
     008E  5E                   STR E             ..(SUM)=D=(ONE)+(TWO)
     008F  E2                   SEX 2             ..RX on stack again
     0090  60                   IRX               ..prime POP, R2=00FE
     0091  72                   LDXA              ..D=M(R2), R2=00FF
     0092  BE                   PHI E             ..RE.1 restored
     0093  F0                   LDX               ..D=M(R2), R2 same
     0094  AE                   PLO E             ..RE.0 restored
     0095  xx xx         ..return, when we learn it
     0097  01            ONE:  DC 1
     0098  02            TWO:  DC 2
     0099  xx            SUM:  ORG *
```

"Indivisible" concerns our operation and events which may occur in RUN mode. RUN mode is a continuous chain of machine cycles regardless of the type (S0, S1, S2 or S3). The operative word is "continuous," and we'll consider that continuity is not destroyed by something like PAUSE mode, since returning directly to RUN mode would pick up exactly where it left off. Time will have elapsed but the logic will continue.

Our operation is whatever we're interested in at the moment; it may be a single instruction, routine, or even an entire program. The operation is indivisible if no event can occur between the start and end of that operation, which will divert ("steal") machine cycles from it. The determining factor is the stolen cycle(s), so PAUSE doesn't count because cycles which do not occur cannot be stolen! If cycles do occur, what can steal them?

Interrupts and DMA's can steal cycles from our program. For either of these, our operation will be suspended at some point, and one or more machine cycles will be stolen to perform another task. If the hardware requesting interrupt or DMA isn't synchronizied with the 1802 cycle, the request will be answered only at a particular time, namely, between machine cycles. Cycles are indivisible. Instructions are also indivisible, so cycles will not be stolen until the instruction in progress is complete. Any operation which requires more than one instruction to complete is divisible.

An event causing division may not occur during the operation in question, but the potential exists unless interrupts are disabled (IE=0) and no DMA requests (which cannot be disabled at the 1802) can occur. Ensuring that no DMA's occur requires that the requesting device be turned off or otherwise prevented from making requests. The 1861 uses both interrupts and DMA, so it will still cause divided operations to refresh the screen if it's on, regardless of whether interrupts are enabled. This is why it is turned off for cassette operations, since those routines are written to use a precise number of machine cycles for timing themselves. Since no routine can tell when or if it has been interrupted, keeping the 1861 on would steal cycles from that carefully counted timing, ruining it.

"Process" is a term which depends upon our perspective and context; like an operation, it may be large or small (to the extremes of involving several programs, machines, and people running them, or observing the activity of a single gate, transistor, or electron!). Process implies activity and a process exists only while it is happenning. A program is only a description of a process. The program on the previous page describes one, two, four, or more processes depending upon how we choose to view it. It may be a single process, assuming that we add a branch at 0007 to get to the ADD routine, and a BR * at 0095 to stop it. We may also consider the creation of the stack to be one process and addition to be a second. We might also think of ADD as the process of saving a register, adding, and restoring the register--three processes. We could even consider each instruction to be a process, but let's not go bananas yet.

If we limit our discussion to two processes, creating a stack and adding, then the process of creating a stack exists only while those 5 instructions are being executed. Adding exists only during the execution of its 20 instructions. Unless we erase memory or load something else, however, the programs describing these processes exist regardless of whether the process does.

Enough confusion for now! I'll use process only to distinguish between our program and interrupt service routines as the only two processes competing for CPU-time. If an interrupt occurs, our process is suspended and the other one then exists.

I make very little distinction between "program," "subprogram," "subroutine," "routine," and "function." I tend to call everything a routine, and most of the time we'll be working on a sub-routine. We'll be turning the ADD routine into a subroutine shortly, and you'll see that there isn't much difference! Most of these terms have arisen as a concession to compatibility between two systems, and often imply a particular protocol employed by the machine code generated by a specific compiler for a high-level language; logically, they all work identically. One routine in a program will NOT be a subroutine--it will be the "main" routine and the starting point from the RUN-switch. All other routines are <u>sub</u>ordinates, relying upon either the main routine or a higher-level subroutine to set them up and to

transfer control to them ("invoke" them). Intuitively, we think of programs as being the largest entity, and as being built of all the other, smaller entities listed, and this concept will suffice.

SUBROUTINES. The ADD routine (0080-0096) is 23 (decimal) bytes long. If we needed to add two numbers in several places in a program, we might just recopy the whole thing into the program at the appropriate points. That seems reasonable, but sooner or later, the routine in question will be a monster (I have at least one subroutine which is 1K long!) and a lot of memory space will be used for redundant code. In that case, we'd need a way to incorporate the LOGIC of that routine into other routines without having to duplicate the code each time. The answer is the subroutine.

To incorporate the logic of one routine within another, we must be able to transfer control of the CPU to the subroutine and regain it when the sub completes. We could simply branch to the ADD routine (with a 30 80 at 0007) and have it branch back to the next instruction at 0009 (30 09 at 0095). If, however, we are at 0020 the next time we need to ADD, we can get to ADD properly, but it will "return" to 0009 (fixed location) instead of our next instruction at 0022. We'd like to regain control wherever we are, and this "return" would eventually get us back to 0020, call ADD, and we'd be in an infinite loop!

Now, we invoke, or "call" the subroutine, relinquishing control to it; we regain control not by any action we can take, but by the subroutine's action of relinquishing control back to us. If the sub does not return, the caller is helpless; the sub is probably in error.

We could get the sub to return to us at different locations by changing the return-branch destination before calling. UGH! Bad idea. Self-modifying code isn't good practise, since we could forget to clean it up again, or forget to set it up first, or if the subroutine is changed, every caller must be changed to make the modification in a new location--easy way to introduce errors. The short branch also means that the caller must be on the same memory page, which is unduly restrictive. It won't work in ROM. And, the caller has to know too much about the sub, which violates the principle of keeping routines "dumb" and independent as far as possible.

We do know, however, that any of the 16 registers may be the PC (R(P), the program counter) as long as a particular register isn't used for two things at once, in conflict. The SEP Technique takes advantage of this feature. We have been using R0 as PC since hitting RUN and haven't changed it yet, so let's continue with R0 as the main-PC. SEP Technique uses the SEP (Dn) instruction to change R(P) from the main-PC to a register used as sub-PC and back again. To call a sub, then, we need a register containing the address of the first instruction of the sub (entry point); then we issue a Dn with the appropriate value of n. We're already using R0 as main-PC, R2 as stack pointer, and RE for work in ADD, so they're not available as the sub-PC; let's use R7. We'll load R7 with the entry address of ADD (0080) and issue a D7 (n=7) to call ADD. Since our PC, as caller, is R0, ADD will have to issue a D0 (n=0) instruction to return to us. R0 will have been left pointing to the instruction following our D7-call, so it needs no adjustment.

```
0007  F8 00 B7          LDI A.1(ADD);PHI 7    ..loading R7
000A  F8 80 A7          LDI A.0(ADD);PLO 7

0095  D0 xx             SEP 0                 ..return from ADD

000D  D7                SEP 7                 ..call ADD
```

This must be simplicity itself! After setting up R7, all we need is a D7 to call ADD every time, as we did at 000D. Well, almost. After the first time, R7 is left at 0096, which is not the entry of ADD, so R7 will have to be reloaded. Every call will have to include all the code shown here at 0007-000D. Let's see if we can improve on "simplicity."

Any register which is the PC when a Dn is executed will be left pointing to the next instruction, and we can use this to "reload" R7 before returning from ADD. We want it pointing to 0080, so let's put the DØ at the location before that (007F), and instead of returning when we get to 0095, we'll branch to 007F instead:

```
007F  D0                SEP 0                 ..return from ADD
0080            ADD:    ORG *
                          :
                          :
0095  30 7F             BR ADD-1              ..reset R7 and return
```

Comparing the ADD routine now with the original version, I think you'll see that there's very little difference between a subroutine and a routine, and why I don't bother with the distinction. We have not made a substantive change.

So, to use SEP Technique, the calling routine must know which register has been set up as the PC for the subroutine (it need not be the same in all cases); the subroutine must know which register is the caller's PC. And, whatever routine sets up the sub-PC (the caller might not do the set-up) must know the entry address of the sub. Let's also distinguish between the "entry" and the "load" address, since ADD's entry is still 0080, but it's load is 007F--the first byte.

SEP Technique has a limitation. The sub must know the caller's PC, but this register ("n" in Dn) is determined when the program is written, and is "bound" (fixed, constant, not variable) at that time. A given sub always returns to the same R(P), and this can be a problem. Suppose we have another routine, DOIT, which uses RD as its PC, and whatever else it does, we're interested in the fact that DOIT calls ADD and DOIT is called by MAIN (our RØ-PC routine). DOIT does not regain control when it calls ADD, since ADD returns to R0, which is where MAIN lives.

```
routine:          MAIN  ADD  MAIN  DOIT  ADD  MAIN
PC:                 0    7     0     D    7     0
nesting level:      0    1     0     1    2     ?
```

As shown in the diagram, ADD does not return to DOIT, so unless calling ADD is the last thing DOIT has to do, it (DOIT) will not finish its job! It also confuses the concept of "nesting level," which indicates how many subroutines have been called without doing a return. We start at level 0, with no calls made, and add 1 to the level for each call and subtract 1 for each return. We would also like to think that we always end up where we started, on level 0. This leads to a conflict with the return from ADD, called by DOIT, to MAIN. Having started with MAIN on level 0, I want to

call this level 0 again; however, being on level 2 (ADD) and doing one return (-1) I get an answer of level 1. In computing, such a conflict means there's an error someplace.

Now, we can get around this problem by having MAIN re-call DOIT with another DD instruction, and RD will be pointing in the right place to allow DOIT to complete. The hexpad software on the VIP does this to get two nibbles to make a byte. We'll just leave the nesting level undefined, but I call that bad practise.

If we knew that ADD returns to R0, we could write DOIT to change its PC to R0, saving the old contents of R0, so the return would be to us. And, if we have a more complex program with more levels, doing this every time is a major production.

Just as "returning" with a branch instruction restricted us to a single location, returning to a single R(P) may also prove to be too restrictive. Just as we improved from a single location to a single R(P), can we also improve to return to variable R(P)'s? YES! With MARK Technique, using the MARK (79) instruction (catchy names, eh?)

To vary the return to different registers depending on which register is the caller's PC, we have to determine which register is used and inform the sub. MARK pushes X and P to the stack as the hi- and lo-order nibbles of a single byte; it also uses R2 explicitly, not R(X) which is why we're using R2 as R(X) as stack pointer! MARK also sets X=P to provide a link to the caller which may be used to pass data to/from the sub. The sub still has its own R(P) and is actually called by SEP, since MARK does not change P. Setting up the sub-PC is the same, but the call is different, than for SEP Technique. The return is also different, since we now have to "SEP" to variable-P, recovering this information from the stack. Both RET (70) and DIS (71) will pop the stack into X and P; RET enables interrupts (IE=1) and DIS disables (IE=0), since they're otherwise the same, I will refer only to RET but everything applies to DIS as well. With the change to the sub, the set-up for its PC will be modified slightly:

```
0007  F8 00 B7          LDI A.1(ADD);PHI 7
000A  F8 7F A7          LDI A.0(ADD);PLO 7

000D              ..the call to ADD
000D  79                MARK                    ..X,P to stack, X=P
000E  D7                SEP 7                   ..call ADD
000F  22                DEC 2                   ..adj. stack ptr.
```

The need for adjusting the stack pointer will be shown in the modification to the sub to return with MARK Technique:

```
007D  12                INC 2                   .."prime" POP
007E  70                RET                     ..X,P from stack, R2+1
007F            ADD: ORG *    ..NOTE: THIS MOVED!
007F  E2                SEX 2                   ..R(X) back on stack
0080            ..ADD routine as before
                         :
                         :
0095  30 7D             BR ADD-2                ..reset R7 and return
```

Both the entry and load addresses of ADD have changed. First, since MARK changed X, we have to change it to 2 so we can save RE on the stack using the existing STXD instructions; the new entry address must be reflected in setting up R7. We've replaced the old return instruction (SEP 0) with RET, which works like LDXA except that X and P are loaded instead of the accumulator. Since this is a POP from the stack, we need the INC 2 to give us the first "advance" step (we could also use IRX). Unfortunately, RET ends with another advance, and you'll recall that we end a POP with a load and no advance (LDX instead of LDXA); so the caller must offset this unwanted advance with the DEC 2 at 000F.

Let's take a look at X, P, R2, and the stack at important points during the call and return with this technique. The values shown are the result after executing the instruction at the location listed:

LOC	X	P	R2	STACK	Notes
0006	2	0	00FF	xx	stack created
000D	0	0	00FE	xx 20	MARK
000E	0	7	00FE	xx 20	SEP 7, R0 left at 000F
007F	2	7	00FE	xx 20	
0083	2	7	00FC	xx (RE.1) (RE.0) 20	RE pushed
0094	2	7	00FE	xx 20	RE popped
007D	2	7	00FF	20	
007E	2	0	0100	xx	R7 left at 007F, NOTE R2!!!!!
000F	2	0	00FF	xx	R2 OK again

We have a lot more freedom once can can save any two registers as X, P of the caller; we are no longer limited to a single "return" location or a single caller R(P). We needed no more set-up than with SEP Technique, and very little more code for calling and in the subroutine. If we manage to leave all the R(P)'s for all the subs we may have unchanged, we need to set up each one only once.

In practise, I've found that with two registers tied up as main PC and the stack pointer, and with the VIP 1861 routine tying up R0, R1, R8, R9 and RB, (only 9 registers left), this seems to get tight in a hurry! With a couple more registers reserved for data (either pointing to it or containing it), I usually felt like I was tripping over myself. Save this PC-value, load another, restore the first, etc. It's probably unfair to call MARK a bad technique, but I haven't used it since learning SCRT. I do wonder if my scheme for register usage could have been better organized to be more harmonious with the hierarchy of routines (who calls whom and when). Then again, most of what I do is "systems," which defies structure and neat hierarchies to a large extent; MARK may look better for non-systems work. I also wonder, if it does take forethought and careful planning to match MARK with a hierarchy for efficiency, whether it is within the capability of beginners. Care in design requires forethought, and that arises from experience, which, by definition, beginners don't have! I guess we could say that forethought from experience is really hindsight from last time.

Anyway. There is a technique which never runs out of PC-registers, called SCRT, and we'll cover that in the next issue.

Dear Ray:

Keep the good work up; I am very much enjoying VIPER again. Paul Piescik's series on machine language is great!

I have a Morse typewriter program which works well. Will send it to anyone interested for 50 cents for copying and a SASE or will send it to VIPER for publication if you think it has general eough interest. Also have a speed adaptive reader program which reads Morse up to 25 words per minute if it's machine-sent, but I'm not satisfied with it yet. Anyone want to exchange ideas?

I don't understand how Football (VIPER 2.05.12) can use high resolution graphics and start at 0200. Perhaps I'm missing a correction?

I have an extra super sound board, VP-550. Anyone know how to convert it to a VP-551? I believe the changes are relatively simple.

73 Don Peterson WB7OCU
2625 E. Southern Ave., C272, Tempe AZ 85282

Well, Don, thanks for the pat on the back. And I do think that there would be interest in your Morse programs. I know that we have a considerable number of amateur radio operators among the membership, and I'm sure that they would all like to see your programs. I've never typed in the Football program, so I can not offer any information on the resolution question.

Converting the VP-550 is simple: on the back (non-component) side of the board, cut the short trace between points "X" and "Y" in the upper right-hand corner. Then install a jumper between pins "K" and "L" (the 9th & 10th from the right edge.) on the edge connector. Pin K has no circuit trace connected to it, so you'll have to solder right on to the gold plating. And that's it! (R.S.)

ANNOUNCEMENTS

This is just a reminder to those VIPHCA members who are also ham radio operators that we have a weekly "meeting" on that air on 75 meters. We usually operate around 3960 KHz at 9 PM local (Eastern) time. Since we have so many members all over the country, perhaps we might consider a net on 20 or 15 meters, so stations at distances greater than a few hundred miles could check in. I'm open to suggestions as to time and frequency. By the way, look for K2ULR (me), K2ZLU George, AE1I Kurt, W2OC Bill, WA2GYM Bill, or KA1IU Mort.

Also, we still have a number of back-issues of VIPER. If you missed any of ARESCO's VIPER issues or would like to replace one or two issues that were lost in the jungle, drop me a line. Volume 1 (partial) is $5 ppd., Volume 2 is $8 ppd. in the U.S. only. Outside of the U.S. please add $2 additional postage for each set. Individual copies are $1.25 each ppd. All payments in U.S. funds. Make your check payable to: VIP Hobby Computer Assn.

READER I/O

Dear Ray:

I enjoyed Tom Swan's CHIP-8 PRIME TIME very much. The little routine in it to shift a keyboard input into MSD will prove very useful in future programs. This allows any entry of data from the keyboard up to FF.

Also, I like the scroll routine. This will be useful in many programs to move data up the screen. I modified the program to continue calculating primes. Even with the scrolling, the routine will calculate and display one thousand primes in 401 seconds. (See Interface Age magazine, August 1981,for the times it takes many popular computers to complete a similar task in BASIC.) Apple took 960 seconds at best, Radio Shack Model I, level II took 1928 seconds. I know this is not a fair comparison, since these were not identical tasks. The routine in Interface Age actually only calculates primes up to the number 1000; however, there is greater difficulty as the primes get larger. The modifications I made are:

```
0228  6A00          026C  7A01          ;This routine will
022A  63FA          026E  3A04          ;calculate 250 primes,
0248  126C          0270  122A          ;4 times around the loop.
                    0272  1202
```

I wonder if anyone has interfaced the Netronics ASCII keyboard to the RCA VIP? I want to use the RCA Tiny BASIC board VP-700, but I would prefer to substitute the Netronics keyboard for the VP-600 series. The thing that may be a problem is the lower case letters. Can the Netronics keyboard be set up for only uppercase letters? Or can it be modified easily to give only upper case? I would rather have the plunger tupe keys than the membrane type of the RCA VP-600 series keyboards. Maybe there is another place to get an inexpensive keyboard, but the Netronics is the only one I have seen. Also, I am assuming that the Tiny BASIC will require only upper case letters. Is this true? I have ordered a tiny BASIC board, but have not received it yet. I don't want to have to hold down the shift key all the time to get upper case letters!

I would appreciate any info on expansion of the VIP in future issues. I am enjoying the newsletter. Thanks for the help.

Gaylord M Ellerman, 2922 69th St., Lubbock TX 79413

OK Gaylord, thanks for the modification to Tom's program. As far as the keyboard for your VIP is concerned, I don't believe that it will be a big problem. The RCA keyboard also generates both upper and lower case letters, and you DO NOT have to hold down the shift key. I am quite certain that there is a software trap in the VP-700 board to remove the possibility of an error being caused by the "wrong" case. By the way, my first "keyboard" for the VP-700 unit was a 8 unit DIP switch and a push button mounted on a PC card and plugged into the I/O port. I toggled in the ASCII codes in BINARY!!!! It sure took a long time to type in a program! (RS)

BOOK REVIEW

Programmer's Guide to the 1802

By Tom Swan

156 pages, softcover.

Hayden Book Company, Inc.
50 Essex Street
Rochelle Park, NJ 07662

$7.95

Those of you who are long-time readers of VIPER will surely recognize the name of Tom Swan. Tom has written much material for the VIPER and has served as VIPER's editor when it was published by ARESCO. Tom's software for the VIP is considered by me and many other people as perhaps the best and most completely documented programs for a computer. And Tom's light and friendly style of writing makes his material very easy to read.

Tom has continued that tradition with his book, Programmer's Guide to the 1802. The book is divided into four large chapters, with several sub-sections in each chapter. At the end of the book there is a "mini-library" of subroutines and an index of the entire book. The first chapter explains the binary and hexadecimal number systems and how mathematical operations are carried out in each. The second chapter, "Fundamentals of Assembly Language" shows how assembly language is used by programmers. Chapter three is a very detailed discussion of the 1802 instruction set, with examples of each instruction type and the mnemonics associated with each 1802 instruction. The last chapter is actually a program: an 1802 assembler. Tom describes in detail how the program is written and what is needed to make it run. By the way, Tom says that he used his VIP with 4K RAM to actually assemble portions of the Assembler! The Assembler is a machine language program and is a commented source listing of the program. It looks very much like the type of listing in Tom's PIPS for VIPS series.

I enjoyed reading this book very much, but perhaps I am too much biased in Tom's favor: I generally like everything he writes. The material would be useful for anyone who would like to venture into assembly language and is certainly worth the $7.95 cover price. The only limitation which might disuade someone from buying the book is that the assembler is not a "ready-to-run" program. You will have to write an input and output driver routine for whatever 1802 machine you have. That might be a problem for someone who is a complete beginner. But if you have written you own programs at all, you should be able to get this one to fly without too much difficulty. After all, you would already have an assembler to help write the output routine, once you get the input routing going! And certainly once you grasp the material in the book, it would be an easy task to write the I/O drivers.

Some VIPHCA members have had a problem locating Tom's book. If you favorite electronics bookstore doesn't have it, try writing the publisher or call them on their toll-free number, 800-631-0856. Lucky New Jersey residents will have to pay for the call plus sales tax, if they order from the publisher.

Raymond C. Sills

VIP Hobby Computer Association
32 Ainsworth Avenue
East Brunswick, NJ 08816

A FINAL WORD:

VIPHCA membership is continuting to grow, with over 200 members on the rolls now. Some of you have probably seen the mention of VIPHCA in the September issue of BYTE magazine. That column has brought us at least a dozen new members. The more we can grow, the better it will be for all of us! Those of you who did not send in the "membership request" form, please let me know if you would like to have your name and address revealed to other VIPHCA members via a membership list. VIPHCA will not divulge this information unless you specifically say it is OK. And we won't sell your name and address to anyone else, either. Don't worry if you didn't send in the membership form: all that the form asked was your name, address, whether you have a VIP or other 1802 machine, and whether you want your address passed to other VIP'ers.

ARITCLES PLANNED FOR COMING ISSUES:

1. CHIP-8 for ELF by Leo Hood
2. CHIP-8 Editor by William Lindley
3. CHIP-8 DO loops by Chuck Rawlins